

Язык программирования Java

Лекция 7

Перевод курса CS11 Java Track
Copyright (C) 2007-2011, California Institute of Technology

Содержание

- Все о потоках Java
- Замечания к заданию 7

Еще раз о потоках в Java

- Программа использует потоки для того чтобы одновременно выполнять несколько задач
 - Поток может иметь собственные локальные ресурсы
 - А так же общие с другими потоками ресурсы
- Для доступа к общим ресурсам должны использоваться атомарные операции
 - Если не следовать этому правилу результат операций будет непредсказуемым
 - Общие ресурсы следует захватывать осторожно, чтобы избежать тупиковых ситуаций и других подобных проблем

Зачем нужны потоки?

- Иногда в отдельных потоках выполняют “медленные” операции
 - Например, передачу данных по сети
 - Во время исполнения такой операции можно продолжать заниматься другими задачами
- Потоки помимо прочего это мощная концептуальная модель построения программного обеспечения
 - Некоторые программы просто проще понять, когда в них используется несколько потоков для решения разных задач
- Использование потоков требует дополнительных (как правило, небольших) ресурсов
 - Процессор должен переключать потоки для того чтобы дать каждому время для исполнения
 - Даже в многоядерной системе тратятся ресурсы на синхронизацию потоков

Задание

- На этой неделе сделаем поисковый робот быстрее!
 - Он тратит много времени на отправку веб запросов и ожидание ответов
- Добавим несколько потоков, в которых будут работать самостоятельные поисковые роботы
 - Каждый будет обрабатывать одну страницу
 - Это позволит нам существенно увеличить производительность программы
 - (... до тех пор пока мы не открыли слишком много потоков)
- Нужно создать “пул ссылок”
 - Роботы будут извлекать ссылки из этого общего пула, и добавлять в него новые найденные ссылки

Пул URL ссылок

- Пул ссылок это общий ресурс
 - Потоки роботов должны использовать атомарные операции для работы с ним
 - Иногда пул может быть пуст
- Как поток должен выполнять атомарные операции надо объектом?
- Как поток должен ждать исполнения условия?

Атомарные операции

- В Java каждый объект имеет монитор
 - Монитор это простой мьютекс (от “mutual exclusion/взаимоисключение”)
 - Объект может быть захвачен одновременно только одним потоком
- Для захвата потока используется блок synchronized

```
synchronized (sharedObj) {  
    ... // выполняем атомарную операцию над объектом  
}
```

- Исполнение потока блокируется до тех пор, пока он не захватит монитор общего объекта
- После этого поток продолжает исполнение
- В конце синхронизируемого блока кода, поток автоматически освобождает монитор объекта

Пример FIFO

- Проблема поставщика и потребителя
 - Один из потоков генерирует данные
 - Другой поток потребляет эти данные
 - Как должны взаимодействовать эти два потока?
- Простое решение : сделать очередь FIFO
 - First In, First Out
 - Поставщик и потребитель оба используют эту очередь
 - Поставщик помещает данные в очередь
 - Потребитель извлекает их оттуда
 - Доступ к очереди из потоков должен быть синхронизирован

Простая очередь FIFO

- Очередь можно сделать с помощью класса LinkedList
- Максимальный размер очереди должен быть ограничен
 - Если поставщик работает быстрее чем потребитель очередь не должна бесконтрольно увеличиваться.
- Класс FIFO:

```
public class FIFO {  
    private int maxSize;  
    private LinkedList items;  
  
    public FIFO(int size) {  
        maxSize = size;  
        items = new LinkedList();  
    }  
    ...  
}
```

Добавление элементов в FIFO

- Если в очереди есть место, добавляем объект и возвращаем true
- Иначе ничего не делаем и возвращаем false:
- Код FIFO:

```
public boolean put(Object obj) {  
    boolean added = false;  
    if (items.size() < maxSize) {  
        items.addLast(obj);  
        added = true;  
    }  
    return added;  
}
```

Извлечение элементов из FIFO

- Если элемент имеется в очереди, извлекаем его и возвращаем на него ссылку
- Если элемент отсутствует в очереди, возвращаем null
- Код FIFO

```
public Object get() {  
    Object item = null;  
    if (items.size() > 0)  
        item = items.removeFirst();  
    return item;  
}
```

Удаление элемента из пустого списка приводит к исключению

Проблема доступа к FIFO из нескольких потоков

- Этот код не годится для использования в разных потоках
 - LinkedList не имеет средств синхронизации доступа к списку из разных потоков, поэтому одновременное добавление и извлечение элемента может привести к непредсказуемому результату
 - Еще хуже ситуация становится в случае, если поставщиков и потребителей несколько
- Пример: имеется два потребителя и один элемент в очереди

```
public Object get() {  
    Object item = null;  
    if (items.size() > 0)  
        item = items.removeFirst();  
    return item;  
}
```

В вызове get оба потребителя вызывая `item.size()` могут получить 1. Это приведет к вызову исключения, когда оба попытаются извлечь последний

Синхронизация операций FIFO

- Для того чтобы таких проблем не было, очередь должна использовать синхронизированные блоки кода

```
public Object get() {  
    public Object get() {  
        Object item = null;  
        synchronized (items) {  
            // этот поток теперь имеет эксклюзивный доступ  
            // к элементам списка.  
            if (items.size() > 0)  
                item = items.removeFirst();  
        }  
        return item;  
    }  
}
```

- Метод put() тоже надо синхронизировать. Операции с элементами списка должны располагаться внутри блока synchronized

Другая проблема FIFO

- Что если в очереди нет данных?
 - Можно сделать цикл, который будет периодически проверять наличие элементов в очереди
 - Этот периодический опрос называется поллинг (англ. polling)

```
//Продолжаем проверять пока не появятся данные!  
do {  
    item = myFifo.get();  
} while (item == null);
```

- Поллинг довольно дорого стоит
 - Он сильно нагружает процессор
 - Всегда следует попытаться найти другое решение

Пассивное ожидание

- Лучше всего если поток пассивно ждет нужное событие
 - Переходит в спящий режим и затем снова просыпается и начинает работать
 - Это можно сделать с помощью методов `wait()` и `notify()`
 - Объявленных в классе `java.lang.Object` (см документацию API)
- После синхронизации на объекте
 - (то есть на мониторе этого объекта)
 - Поток может вызывать метод `wait()`, который переводит его в спящее состояние
 - Поток освобождает монитор перед переходом в спящее состояние
- Объект можно “ждать” только после синхронизации на нем
 - Иначе будет вызвано исключение `IllegalMonitorStateException`

Выход из спящего состояния

- Из спящего состояния поток может быть выведен другим потоком
 - Для этого другой поток должен синхронизироваться на объекте
 - Затем вызвать метод `notify()` или `notifyAll()` для того чтобы возобновить работу всех потоков ждущих этот объект
 - Если таких ждущих потоков нет, то при вызове `notify()` или `notifyAll()` ничего не происходит
- Эти методы можно вызывать у объектов только после синхронизации над ними ...

Уведомления

- Когда поток получает сообщение, он немедленно пытается захватить объект на котором он вызвал `wait()`
 - Функция `wait()` вызывается внутри блока `synchronize...`
 - Но поток, который вызвал `notify()` по прежнему владеет блоком
- Когда поток, отправляющий уведомление освобождает объект, один из потоков получивших уведомление, захватывает блок в свою очередь.
 - Следующий поток произвольно выбирается виртуальной машиной Java
 - Выбранный поток возобновляет исполнение и получает исключительные права на объект синхронизации

Как использовать `wait()` и `notify()`

- Стандартный сценарий выглядит так:
 - Один из потоков не может продолжать работу, пока не выполнится какое то условие
 - Поток вызывает `wait()` и переходит в состояние ожидания
 - Метод очереди `get()` может вызвать `wait()`, если очередь пуста
- Другой поток изменяет состояние объекта
 - Ему известно о том что условие выполнено
 - Он вызывает `notify()` или `notifyAll()` для того чтобы возобновить работу всех ожидающих события потоков
 - Метод очереди `put()` может вызвать `notify()` когда в очередь добавляется новый элемент


Как использовать wait()

- Поток, ожидающий событие не должен по определению считать что событие, которое он ждал, произошло
 - Если несколько потоков ждали, используя один и тот же объект синхронизации, и был вызван метод notifyAll, событие для обработки могло быть передано сначала другому потоку
- Можно использовать wait() с таймаутом
 - Тогда метод возвращает управление, если приходит сообщение о событии или если истекает заданный интервал времени
- Также возможны “ложные” срабатывания
 - Поток просыпается без уведомления
 - Это возможно в некоторых реализациях JVM
- Из всего этого следует, что wait() надо использовать в цикле, дополнительно проверяя условие после того как эта функция вернет управление

Вернемся к FIFO

Теперь используем метод `wait()` для работы с очередью:

```
public Object get() {  
    Object item = null;  
    synchronized (items) {  
        // Этот поток имеет эксклюзивные права доступа  
        // к элементам очереди  
  
        // Ждем пока в очереди не появится элемент  
        while (items.size() == 0)  
            items.wait();  
  
        item = items.removeFirst()  
    }  
    return item;  
}
```




Всегда ждем в цикле, который проверяет условие

Вывод потребителя из состояния ожидания

- Метод put() используется для того чтобы отправить потребителям уведомление о появлении нового элемента в очереди:

```
public boolean put(Object obj) {  
    boolean added = false;  
    synchronized (items) {  
        if (items.size() < maxSize) {  
            items.addLast(obj);  
            added = true;  
            // элемент добавлен, надо разбудить потребителей.  
            items.notify();  
        }  
    }  
    return added;  
}
```



Вызываем notify() на том же объекте,
который ждут другие потоки

Еще одна проблема...

- Если поставщик работает быстрее чем потребитель, он не может ждать пока в очереди появится свободное место!
 - Потребитель может ждать, но ...
 - Поставщик может только опрашивать очередь, чтобы выяснить, что в ней нет места
- Наша реализация в действительности очень проста 😊
 - В ней имеются и другие проблемы!
 - Например, один и тот же объект используется для синхронизации методов `get()` и `put()`
- Классы пакета `java.util.concurrent` содержат сложные реализации очередей, пулов и других структур
 - Они добавлены в версию Java 1.5! Эти классы написаны Дугласом Ли

Синхронизация по this

- Объект можно синхронизировать на себе самом
 - Это особенно удобно, если объект используется для управления несколькими общими ресурсами
 - Надо быть осторожным – захват нескольких ресурсов вручную может привести к тупиковой ситуации
- Вместо захвата items очередь может делать так:

```
public Object get() {  
    Object item = null;  
    // Захватываю свой собственный монитор.  
    synchronized (this) {  
        while (items.size() == 0)  
            wait(); // Вызываю wait() на самом себе.  
        item = items.removeFirst();  
    }  
    return item;  
}
```

Синхронизированные методы

- Синхронизация на `this` применяется очень часто
- В Java для это есть другой синтаксис:

```
public synchronized Object get() {  
    while (items.size() == 0)  
        wait();  
    return items.removeFirst();  
}
```

- `this` захватывается в начале метода
- `this` освобождается в конце метода
- Внутри метода можно вызывать `wait()` и `notify()`
- Чтобы сделать класс “безопасным” для работы в многопоточном приложении достаточно добавить модификатор `synchronized` ко всем его методам
 - (кроме конструкторов, для которых это просто не нужно)

Потоки и производительность

- На синхронизацию потоков тратятся системные ресурсы
 - Захват и освобождение мьютекса занимает время
 - Не следует использовать синхронизацию в случаях, когда без этого можно обойтись
- Плохие примеры:
 - `java.util.Vector`, `java.util.Hashtable`
 - Оба эти класса синхронизируют каждый свой метод!
 - Не используйте их в приложениях с одним потоком (или, может быть, вообще нигде не используйте?)
- Потоки должны захватывать общие ресурсы на минимально короткое время
 - Старайтесь свести к минимуму взаимодействие потоков друг с другом

Замечания к заданию 7

- Нужно создать пул объектов URLDepthPair
 - Этот пул будет общим для всех потоков поискового робота
 - Потоки робота извлекают ссылки из пула и добавляют в него новые найденные ссылки
- Внутренние поля:
 - Один экземпляр класса LinkedList нужен для хранения ссылок, которые должны быть просмотрены
 - Другой экземпляр класса LinkedList должен хранить уже просмотренные ссылки
- Методы:
 - Метод , возвращающий следующий экземпляр класса URLDepthPair для обработки
 - Поток должен переходить в состояние ожидания, если в пуле отсутствуют объекты
- Метод , добавляющий в пул новый URLDepthPair для обработки
 - Новый элемент всегда следует добавлять к списку просмотренных ссылок
 - Если его глубина меньше максимальной, то еще и к списку ссылок, для просмотра
 - Если элемент добавляется к списку ссылок для просмотра, то дополнительно надо отправить сообщение потокам, ждущим заполнения пула

Наиболее сложная задача

- Когда завершается поиск? Как мы можем это узнать?
 - Когда все потоки поискового робота переходят в состояние ожидания, работа закончена
 - (Хорошо чтобы еще очередь ссылок на просмотр была пуста!)
- Пул ссылок должен иметь счетчик ждущих потоков
 - Это легко сделать:
 - В конструкторе присвойте счетчику значение 0
 - Инкрементируйте значение счетчика перед вызовом `wait()`
 - Декрементируйте счетчик после того как `wait()` вернет управление
- Основной поток приложения может периодически проверять этот счетчик
 - Ему известно, сколько потоков робота занято
 - В конце работы, так или иначе, ему надо напечатать на экране результат работы
 - Позаботьтесь о том, чтобы синхронизировать доступ к этому счетчику

Потоки работа

- Создайте класс CrawlerTask, который реализует интерфейс Runnable
 - В CrawlerTask нужна ссылка на URLPool
 - Подсказка: передайте ссылку на URLPool в конструкторе CrawlerTask
 - Метод run() должен содержать цикл в котором:
 - Из пула извлекается URL
 - Загружается страница и в ней ищутся новые ссылки
 - Найденные новые ссылки помещаются обратно в пул.
 - Возвращаемся к началу цикла
 - Каждая ссылка должна обрабатываться в вспомогательном методе (или нескольких методах)
 - Используйте код из прошлого задания
 - Правильно обрабатывайте исключения
 - Если при обработке одной ссылки произошла ошибка, переходите к обработке следующей

Метод `main` робота

- Метод `main()` управляет всем процессом от начала до конца
 - Получает начальную `http` ссылку, максимальную глубину и количество потоков из параметров командной строки
 - Создает пул ссылок и добавляет в него начальную ссылку
 - Создает и запускает заданное число потоков
 - Их можно поместить в массив, чтобы впоследствии удалить, но в этом задании делать это не обязательно
 - Проверяет пул с периодом 0.1 – 1 секунды, чтобы выяснить завершена работа или нет
 - Если работа завершена, печатает ссылки из списка просмотренных URL
 - Вызывает `System.exit(0);`

Работа с потоками

- Для создания потока надо создать класс реализующий интерфейс Runnable
 - Добавить свою реализацию метода run()
- Экземпляр этого класса надо указать в аргументах конструктора класса Thread:

```
CrawlerTask c = new CrawlerTask(pool);  
Thread t = new Thread(c);
```

- Вызовите метод start() созданного объекта Thread

```
t.start();
```

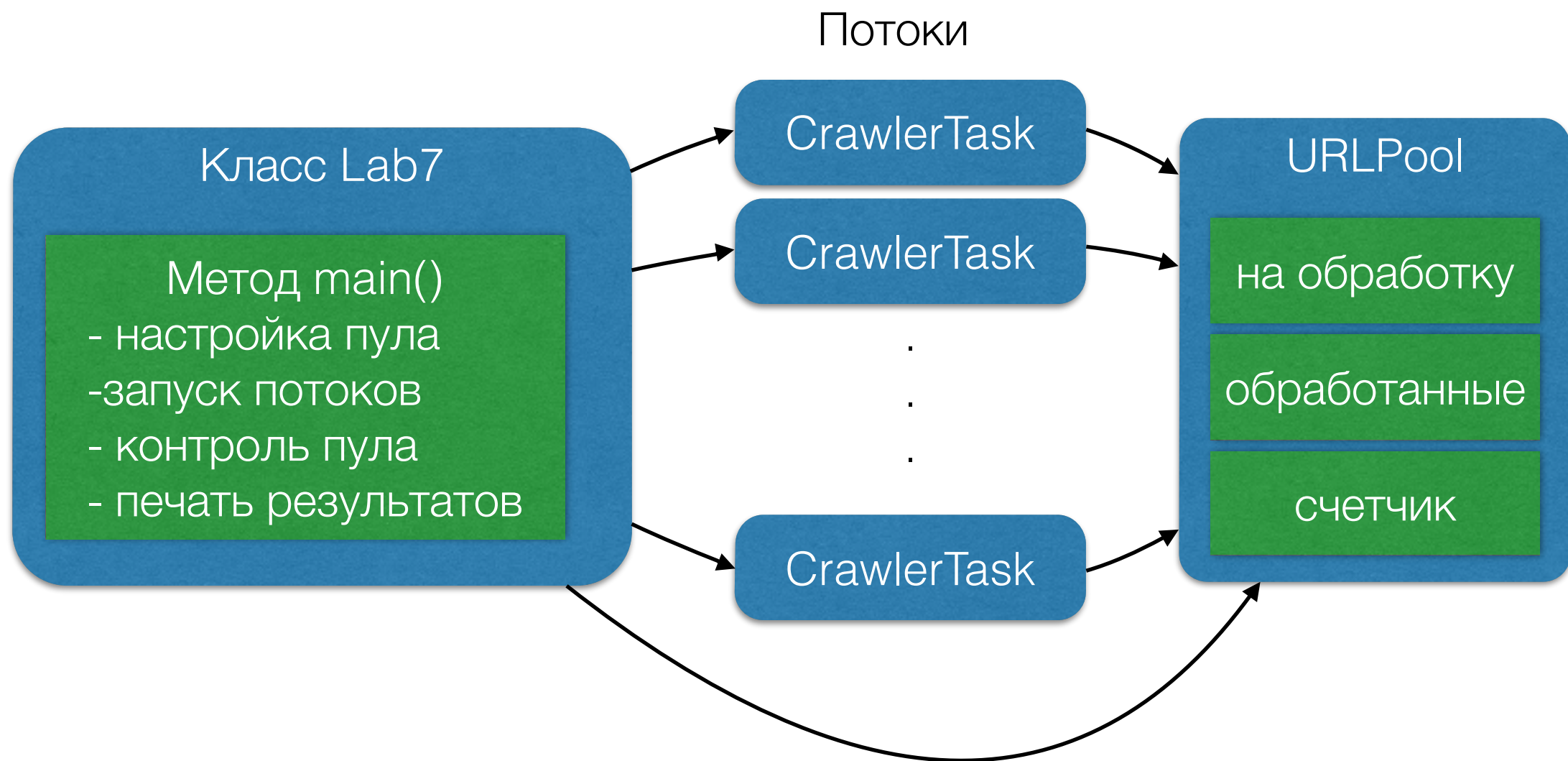
- Поток автоматически вызовет метод run() вашего объекта
- Поток завершит работу, когда завершится исполнение метода run();

Аккуратный опрос

- Используйте метод `Thread.sleep()` для вставки пауз между проверками
 - `sleep()` это статический метод
 - Может вызывать исключение `InterruptedException`!
 - Пожалуй, лучший способ периодического опроса
- Делается как то так:

```
while (pool.getWaitCount() != numThreads) {  
    try {  
        Thread.sleep(100); // 0.1 second  
    } catch (InterruptedException ie) {  
        System.out.println("Ошибка " +  
            "InterruptedException, игнорируем...");  
    }  
}
```

Общая картина



Синхронизация пула

- URLPool содержит несколько общих ресурсов:
 - Список ссылок для просмотра, список просмотренных ссылок, счетчик потоков ожидающих данные для обработки, ...
- URLPool может синхронизироваться на самом себе
 - Это помогает избежать тупиковых ситуаций и других подобных проблем
- URLPool должен иметь внутреннюю поддержку синхронизации потоков
 - Потоки поискового робота не должны самостоятельно “вручную” выполнять операции `synchronize/wait/notify` над пулом
 - Поведение потоков хочется также скрыть внутри пула

Литература по потокам Java

- Doug Lea, Concurrent Programming in Java (2nd ed.)
- Joshua Bloch, Effective Java